深層学習入門

深層学習に入門してみた

Hiroshi Hatake

Technical information sharing seminar

はじめに

機械学習とは

機械学習とは、あるデータセットから特徴量と呼ばれる定量化された指標を抽出し、それらの値を元に未知の入力値から値を取り出すことである。

はじめに

噛み砕くと

あるデータの傾向を繰り返しの計算によって予測し、その予測のパターンに未知の値を当てはめる行為。

はじめに

機械学習の流れ

機械学習の流れ

機械学習の流れ

機械学習の流れ

- ある既知のデータの集まりを取得する。

機械学習の流れ

機械学習の流れ

- ある既知のデータの集まりを取得する。
- 既知のデータの集まりを扱いやすい形に加工する。

はじめに

機械学習の流れ

機械学習の流れ

- ある既知のデータの集まりを取得する。
- 既知のデータの集まりを扱いやすい形に加工する。
- 取得したデータの集まりを元に入力値の傾向を反復計算する。

はじめに

機械学習の流れ

機械学習の流れ

- ある既知のデータの集まりを取得する。
- 既知のデータの集まりを扱いやすい形に加工する。
- 取得したデータの集まりを元に入力値の傾向を反復計算する。
- 計算した入力値の傾向を元に、未知の入力値から値を取り出す。

背景

機械学習の手法

主にどんな種類のものがあるのか

- 教師あり学習
- 教師なし学習
- 教師ありなし混合学習
- etc.

深層学習 (Deep Learning)

深層学習

深層学習とは入力の傾向の特徴を学習データから自動で抽出する

深層学習 (Deep Learning)

深層学習と機械学習の違い

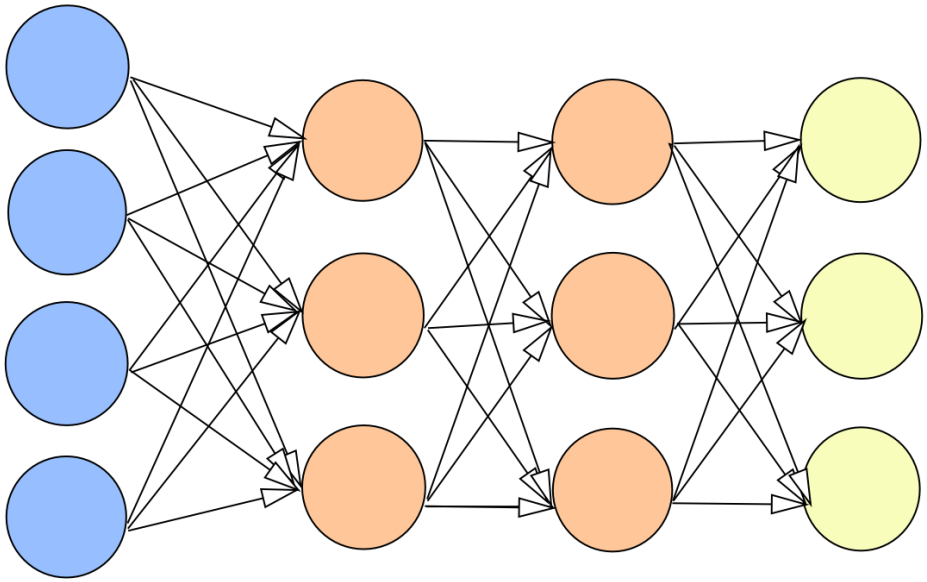
- 機械学習では予測に用いる傾向の選び方を人間が決める必要がある

深層学習 (Deep Learning)

深層学習と機械学習の違い

- 機械学習では予測に用いる傾向の選び方を人間が決める必要がある
- 深層学習では予測に用いる傾向の選び方を自動で獲得する

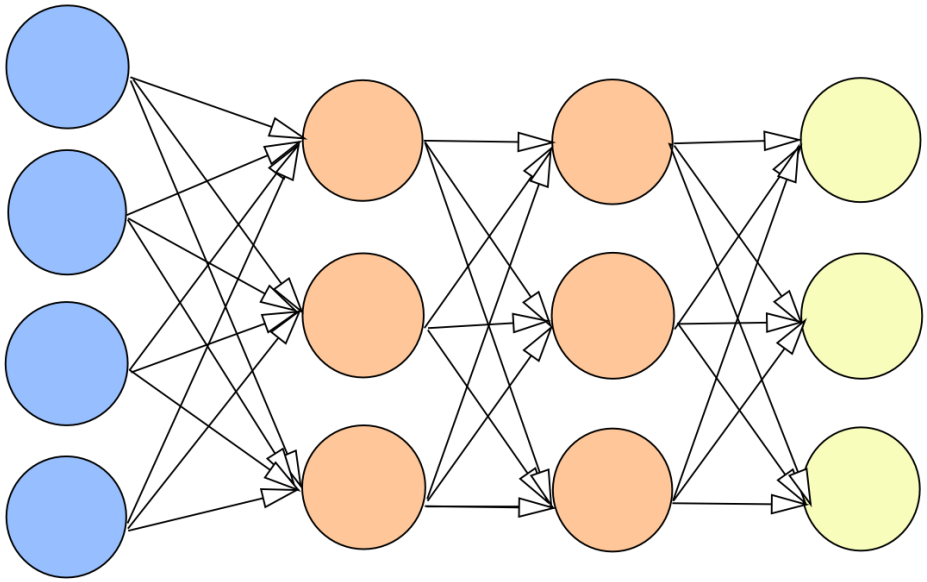
深層学習の解説の前に



ニューラルネットとは

- 値が伝わっていく向きのあるグラフ

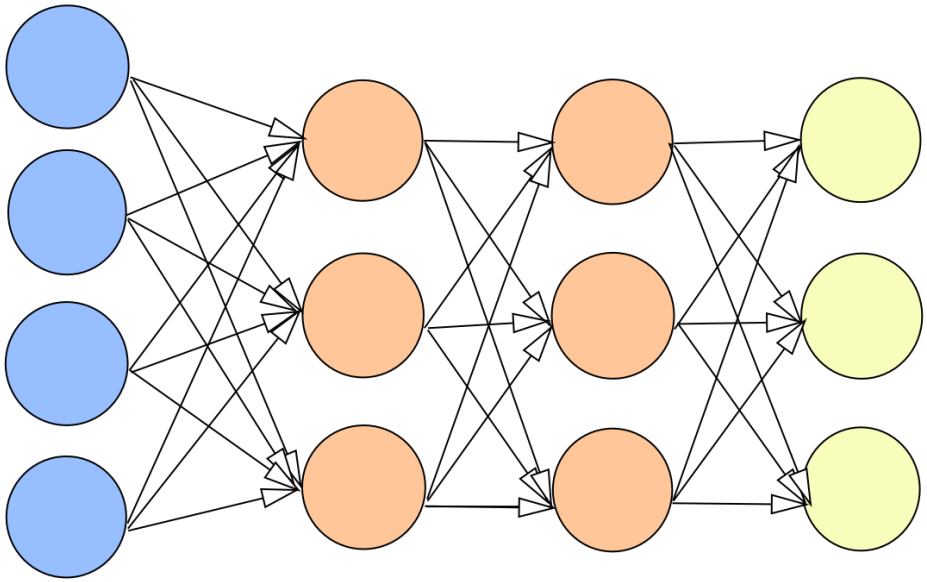
深層学習の解説の前に



ニューラルネットとは

- 値が伝わっていく向きのあるグラフ
- グラフの層ごとに重み付けをしたり変換をしたり

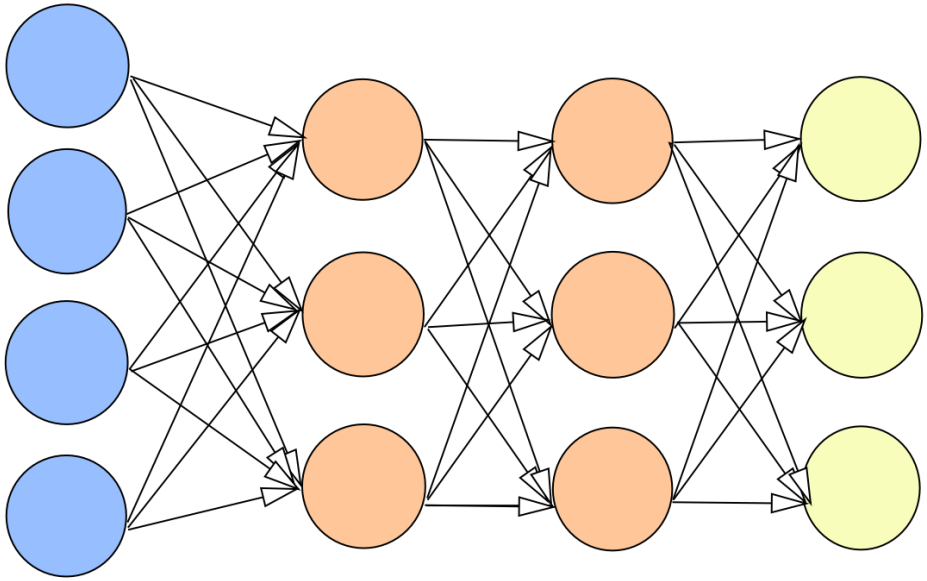
深層学習の解説の前に



ニューラルネットとは

- 値が伝わっていく向きのあるグラフ
- グラフの層ごとに重み付けをしたり変換をしたり
- グラフ全体を見ると一個の巨大な関数を組み合わせたもの（合成関数とも呼ばれます）

深層学習の解説の前に



このニューラルネットと呼ばれるものを使い、未知の入力値を変換し、目的の値を取り出す。[1]

[1]実際にはこのニューラルネットを使う目的関数の設計と勾配の計算、計算したモデルの誤差を定めた誤差関数を最小化する反復計算の手順を踏む。このスライドでは深入りしない。

深層学習の適用範囲

応用範囲

- 画像認識

深層学習の適用範囲

応用範囲

- 画像認識
- 音声認識

深層学習の適用範囲

応用範囲

- 画像認識
- 音声認識
- 言語処理

深層学習の適用範囲

応用範囲

- 画像認識
- 音声認識
- 言語処理

どれも非常にデータ量が多くなるタスク

深層学習のフレームワーク

フレームワーク

- Caffe[2]
- Torch[3]
- TensorFlow[4]
- Chainer[5]

[2] http://caffe.berkeleyvision.org/
[3] http://torch.ch/
[4] https://www.tensorflow.org/
[5] http://chainer.org/

Chainer

Chainer とは

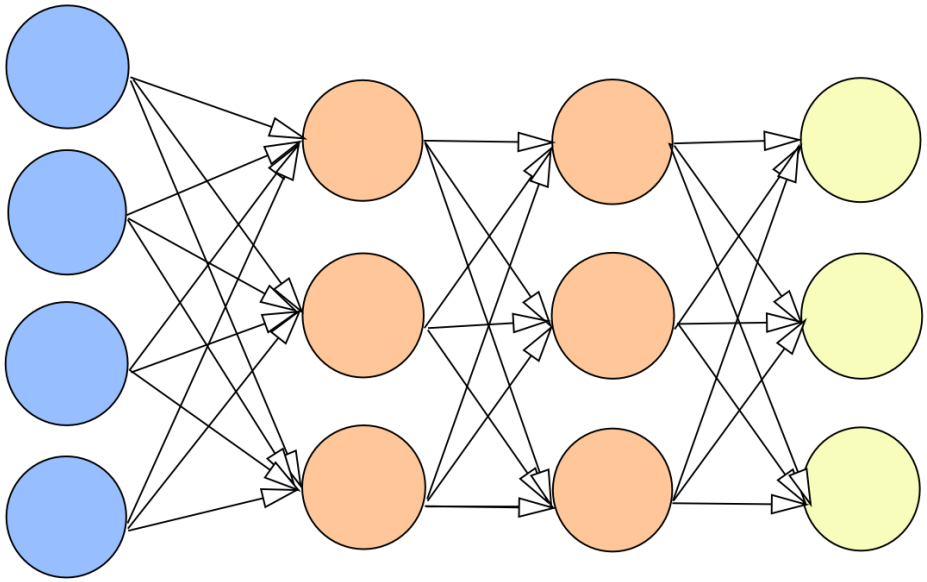
- CUDA サポートがあり、計算に GPU が使用可能
- ニューラルネットの設計が Python の DSL で行え、高い柔軟性がある
- 値の伝搬のコードだけ書けば、逆誤差伝搬法 (backpropagation)[6] は自動で計算する

[6]勾配を求める手法で、この方法を用いると計算コストを抑えた効率的な計算ができることが知られている。
http://neuralnetworksanddeeplearning.com/chap2.html などを参照のこと。

Chainer

Chainer の特徴

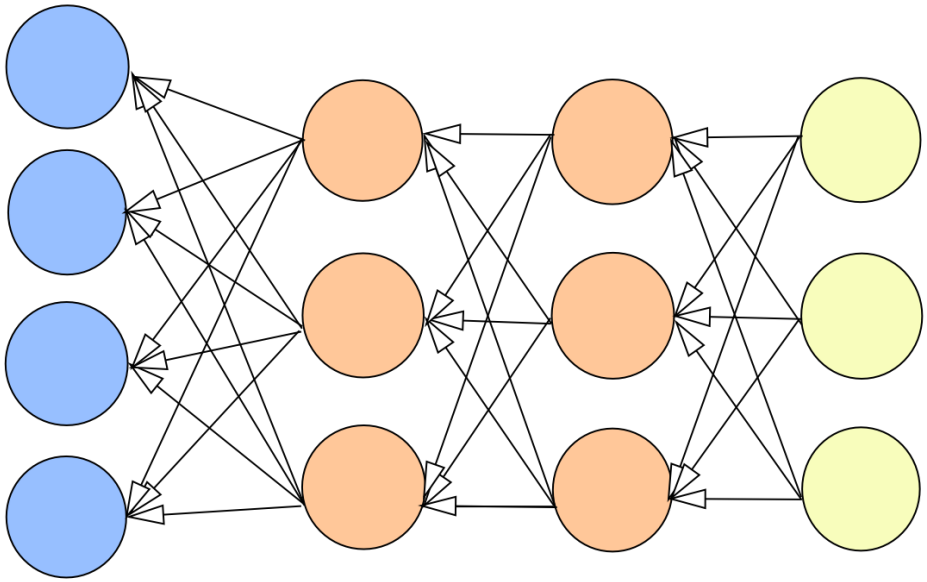
ニューラルネットの設計。ここは書く必要がある。



Chainer

Chainer の特徴

backpropagation。ここは自動で計算してくれる。



Chainer

Chainer の特徴

CUDA サポートがある。CUDA とは NVIDIA の GPU の汎用計算環境で、GPU を計算目的に使用することができる。[7]

[7]実際は Chainer では CuPy として NumPy のインターフェースに則ったライブラリから CUDA を使用するようになっている。詳細は http://www.slideshare.net/ryokuta/cupy などを参照のこと。

Chainer を動かしてみる

Chainer を動作させた環境

- OS Ubuntu 14.04.4 LTS
- Mem 32GB
- GPU GTX 1070 VRAM 8GB
- CUDA 8.0RC
- cuDNN 5.0
- chainer 1.10

応用例1

画風を変換するアルゴリズム

応用例として画風を変換するアルゴリズムがある。“A Neural Algorithm of Artistic Stlye”[8] このスライドではVGGネットワーク[9]を用いた結果を提示する。

[8] http://arxiv.org/abs/1508.06576
[9] http://arxiv.org/pdf/1409.1556

応用例1

content loss

$$\mathcal{L}_{content}(\vec{p}, \vec{x}, l) = \frac{1}{2} \sum_{i,j} (F_{ij}^l - P_{ij}^l)^2 \tag{1}$$

style loss

$$G_{ij}^l = \sum_k F_{ik}^l F_{jk}^l \tag{2}$$

$$E_l = \frac{1}{4N_l^2 M_l^2} \sum_{i,j} (G_{ij}^l A_{ij}^l)^2 \tag{3}$$

$$\mathcal{L}_{style}(\vec{a}, \vec{x}) = \sum_{l=0}^{L} w_l E_l \tag{4}$$

loss

$$\mathcal{L}_{total}(\vec{p}, \vec{a}, \vec{x}) = \alpha \mathcal{L}_{content}(\vec{p}, \vec{x}) + \beta \mathcal{L}_{style}(\vec{a}, \vec{x}) \tag{5}$$

応用例1

画風を変換するアルゴリズム

スタイル画像とスタイルを適用したい画像を用意する。[10]


・・・→?

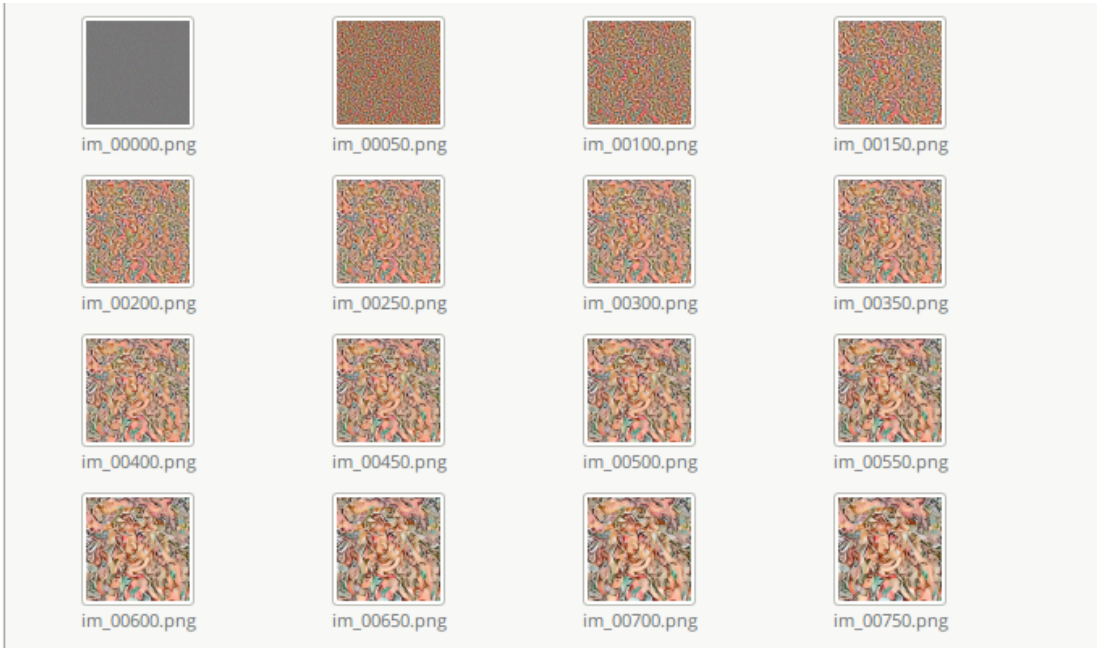
[10] https://github.com/mattya/chainer-gogh に従って画像のスタイル適用を試しています。

応用例1

Chainer

画風を変換するアルゴリズム

どんどんと誤差関数が小さくなるように計算を回していくと・・・？



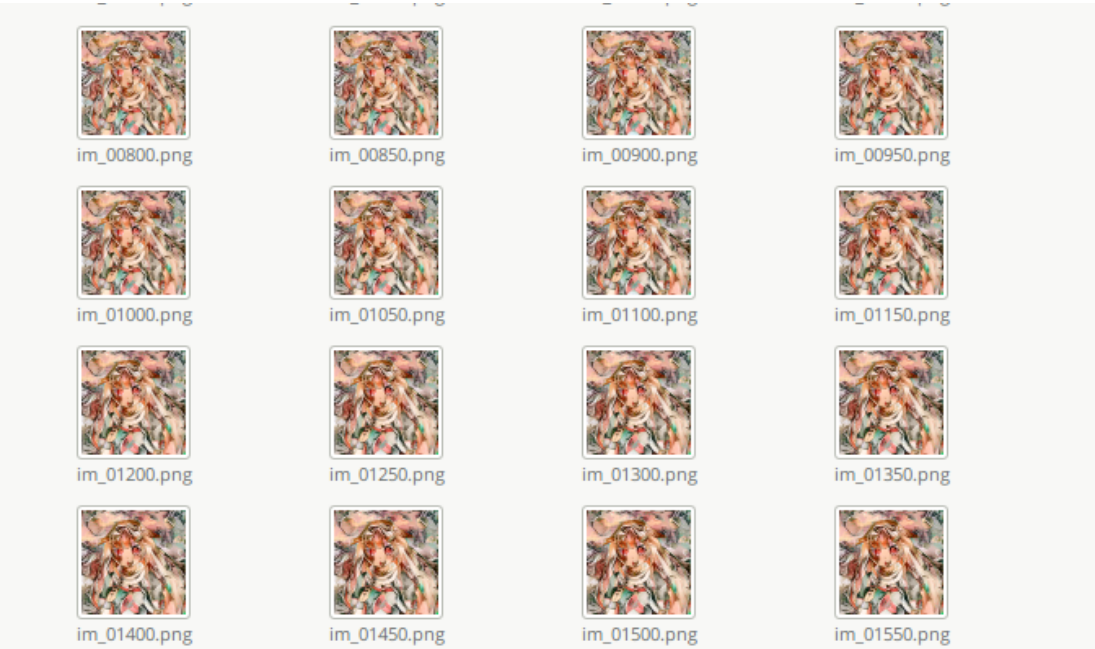
・・・→?

応用例1

Chainer

画風を変換するアルゴリズム

どんどんと誤差関数が小さくなるように計算を回していくと・・・？



・・・→?

応用例1

画風を変換するアルゴリズム

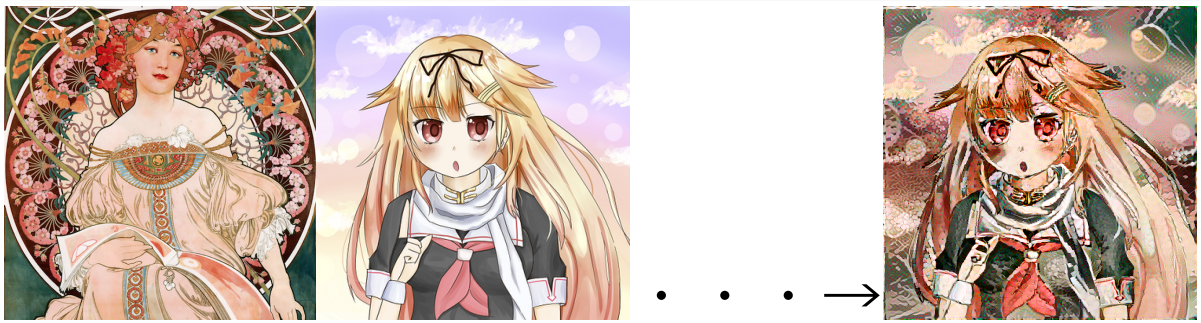
どんどんと誤差関数が小さくなるように計算を回していくと・・・？

応用例1 – 結果

Chainer

画風を変換するアルゴリズム

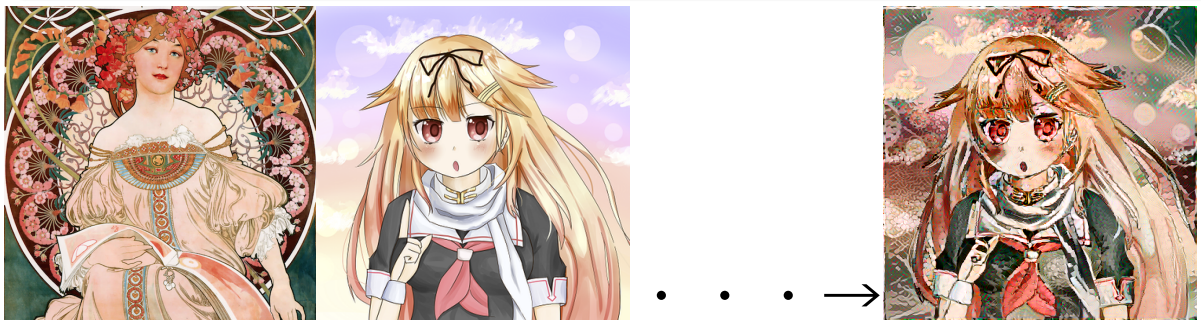
スタイル画像と入力画像、スタイル適用後の画像。掛かった時間は 313.85 秒。パラメータを初期値のまま走らせると VRAM を 2.5GB ほど持って行きます。VRAM 不足に注意。



応用例1 – 結果

画風を変換するアルゴリズム

確かにそれっぽい画像を出力してくれる。…が、GTX 1070 で VGG ネットワークを用い、CUDA で計算をしても最終的な出力が得られるまで 5 分強の時間が掛かってしまう。



応用例1

スタイル適用に時間かかりすぎ！

応用例1

スタイル適用だけを速くしたい！

応用例2

画風を変換するアルゴリズムその2

応用例として画風を変換するアルゴリズムのその2。“Perceptual Losses for Real-Time Style Transfer and Super-Resolution”[11] このスライドではVGGネットワーク[12]を用いた結果を提示する。[13]

---

[11] http://arxiv.org/abs/1603.08155

[12] http://arxiv.org/pdf/1409.1556

[13] https://github.com/yusuketomoto/chainer-fast-neuralstyle に従って画像のスタイル適用を試しています。

応用例2

feature reconstruction loss(content loss)

$$l_{feat}^{\phi,j}(\hat{y}, y) = \frac{1}{C_j H_j W_j} \|\phi_j(\hat{y}) - \phi_j(y)\|_2^2 \tag{6}$$

style reconstruction loss(style loss)

$$G_j^{\phi}(x)_{c,c'} = \frac{1}{C_j H_j W_j} \sum_{h=1}^{H_j} \sum_{w=1}^{W_j} \phi_j(x)_{h,w,c} \phi_j(x)_{h,w,c'} \tag{7}$$

$$l_{style}^{\phi,j}(\hat{y}, y) = \| G_j^{\phi}(\hat{y}) - G_j^{\phi}(y) \|_F^2 \tag{8}$$

total loss

$$\hat{y} = \arg\min_y \lambda_c l_{feat}^{\phi,j}(y, y_c) + \lambda_s l_{style}^{\phi,J}(y, y_s) + \lambda_{TV} l_{TV}(y) \tag{9}$$

応用例2

画風を変換するアルゴリズムその2

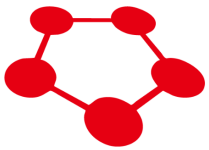
まずはVGGネットに学習をさせてスタイルを適用するニューラルネットを作成する。また、学習にはMicrosoft COCO dataset[14]を用いた。

ソースコード 1: train with style image and dataset

```
$ python train.py -s image/style_6.png -d train2014 \
  -g 0 -o style6
```

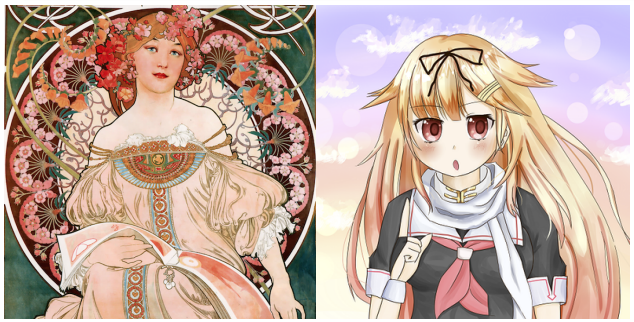
[14] http://mscoco.org/dataset/#download

応用例2



画風を変換するアルゴリズムその2

まずはVGGネットに学習をさせてスタイルを適用するニューラルネットを作成する。



・・・→?

応用例2

画風を変換するアルゴリズムその2

Microsoft COCO dataset は約8万枚の13GBにもなる画像セット。これを元にスタイルを適用するニューラルネットを作成する。環境は、先ほどと同一。GTX 1070で2回学習ループを回すのにおよそ4・5時間ほどかかる。[15]

[15] はずなのだが、正確な時間を取り忘れた。なぜなら学習にかけてそのまま寝て起きたら終わっていたから。

応用例2

実際に画風を変換するニューラルネットを適用してみる

学習済みニューラルネットを適用するのはほぼ一瞬。

ソースコード 2: apply trained neural net

```
$ python generate.py sample_image/yuudachi_400x400.png \
  -m models/style6.model \
  -o sample_image/yuudachi_style6.png -g 0
0.949225902557 sec
```

応用例2

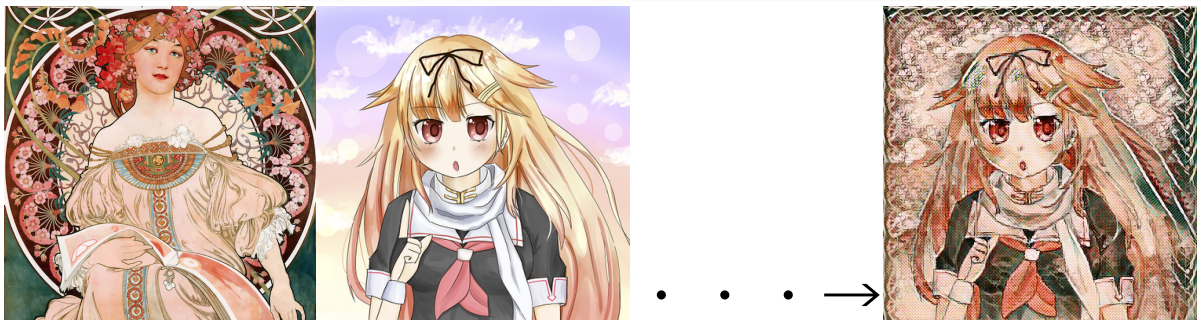
0.949225902557 sec

応用例2 – 結果

Chainer

画風を変換するアルゴリズムその2

確かにそれっぽい画像を出力してくれる。今回は学習済みのモデルを使えば出力が格段に速い。応用例1比で 313.85/0.49 ≒ 640.5 倍の速さ。およそ3桁倍高速化をしている。（論文中でも “up to three orders of magnitude faster” との記述が！）



まとめ

- 学習するときはかなりの計算コストがかかる。
- このスライドでは省略している箇所がありますが、計算の裏付けの数式があります。[16]
- 定番のニューラルネットをダウンロードしてきて Chainer で使える。
- 実は Caffe などで作成したモデルも Chainer で読み込める！
- 学習済みの結果を使うときは（使い方にもよりますが）、普通のアルゴリズムと同じように使えます。
- Chainer はいいぞ。

[16] http://bookclub.kodansha.co.jp/product?isbn=9784061529021 がよくまとまっていて詳しい